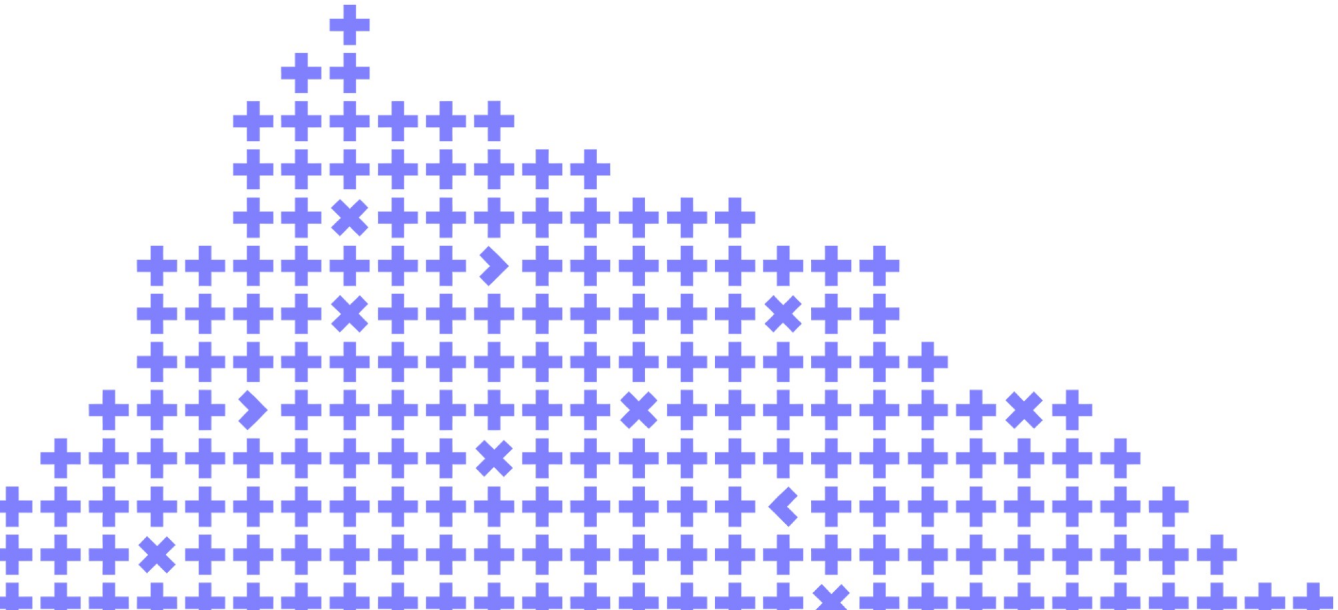
World Constants ‘2022

Andrew Aksyonoff //  //  // v.0.5

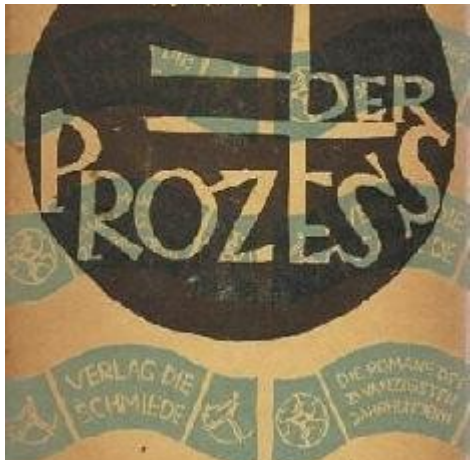


Co-organizer

Yandex

Bio, right?

- Who’s here for the Kafka talk then?



Bio, right?

- Who’s here for the Kafka talk then?
- Who really cares about bios?
- **Why** do you care again?

Bio, right?

- Who’s here for the Kafka talk then?
- Who really cares about the bios?
- **Why** do you care about them?

- AOL, AltaVista, Sphinx, Samsung, Fed Treasury, Google, Craigslist, Valve, P***Hub, Inktomi, Salesforce, EPAM, 1C, Apple, Avito, Amazon, Ozon, ...
- Pick a few you like then

let’s start backwards

- This will probably be a **BAD** talk... ;(
- ...hopefully inciting some **GOOD** thinking!

- Do you know the **required** “world **CONSTANTS**”?
- Do you know the **CURRENT** ones?
- Let’s **BRIEFLY** cover all that jazz
- This will probably be a **BAD** talk… ;(
- …hopefully inciting some **GOOD** thinking!

- Can you **COUNT**? As in…
- Can you **PREDICT** the system performance?
- Do you know the **required** “world **CONSTANTS**”?
- Do you know the **CURRENT** ones?
- Let’s **BRIEFLY** cover all that jazz (cf 112)
- This will probably be a **BAD** talk… ;(
- …hopefully inciting some **GOOD** thinking!

“World constants?!”

e

$$\sqrt{-1}$$

but, IT version?

zee principal intermezzo

[Dean2010]

L1 cache reference	0.5 ns	/
Branch mispredict	5 ns	/
L2 cache reference	7 ns	/ 14x L1 cache
Mutex lock/unlock	25 ns	/
Main memory reference	100 ns	/ 20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	3,000 ns	/
Send 1K bytes over 1 Gbps network	10,000 ns	/
Read 4K randomly from SSD*	150,000 ns	/ ~1GB/sec SSD
Read 1 MB sequentially from memory	250,000 ns	/
Round trip within same datacenter	500,000 ns	/
Read 1 MB sequentially from SSD*	1,000,000 ns	/ ~1GB/sec SSD, 4X memory
Disk seek	10,000,000 ns	/ 20x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000 ns	/ 80x memory, 20X SSD
Send packet CA->Netherlands->CA	150,000,000 ns	/

Alas, things decay

- That was then – **2010**, this is now – almost **2023**
- [Dean2010] is still a **GREAT** start, but

- Today it’s (**UPTO**) **SOMEWHAT OFF**
- Anyday it’s **NOT QUITE COMPLETE**
- Today I have **UPDATES & ADDITIONS**
- And, of course, benchmarks

the 3 cornerstones?

1

CPU

2

RAM

disk

3

let’s start easy!

Disk!

- A single “IO” => reading a certain “atom” (sector)
- Sectors are 4096 bytes now

- HDD = **~100…200** random iops total
- HDD = **~100…200 MB/sec** linear… no change there, but

- SSD = **~500K (!!!)** total 32T iops, **~50K** 1T iops (evo 970)
- SSD = **~3500…5000+ mb/sec linear**

Disk!

- This actually ALREADY differs vs scripture...

- Smaller thing – newer hardware, faster reads
- Bigger thing – **UNITS**

[Dean2010]

L1 cache reference	0.5 ns	/
Branch mispredict	5 ns	/
L2 cache reference	7 ns	/ 14x L1 cache
Mutex lock/unlock	25 ns	/
Main memory reference	100 ns	/ 20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	3,000 ns	/
Send 1K bytes over 1 Gbps network	10,000 ns	/
Read 4K randomly from SSD*	**150,000 ns**	**/ ~1GB/sec SSD**
Read 1 MB sequentially from memory	250,000 ns	/
Round trip within same datacenter	500,000 ns	/
Read 1 MB sequentially from SSD*	**1,000,000 ns**	**/ ~1GB/sec SSD, 4X memory**
Disk seek	**10,000,000 ns**	**/ 20x datacenter roundtrip**
Read 1 MB sequentially from disk	**20,000,000 ns**	**/ 80x memory, 20X SSD**
Send packet CA->Netherlands->CA	150,000,000 ns	/

Disk!

- **“100 MB/sec”** vs **“10…20,000,000 ns/MB”**?!

- Personally, MB/sec, ofc, any day
- Meh option, **msec**/MB

- But, pick **YOUR** poison

moving on

CPU!

- Refresher, a few general facts

- CPU reads both data **and** code from **RAM**
- CPU **caches** those reads, multi-layer **L1** / **L2** / etc
- CPU then works with internal **REGISTERS**
- And does so in “regular” **CLOCK TICKS**
- 1 physical multi-core CPU == N logical + shared RAM

CPU!

- Fun fact, **ALL** those are just “tips of the icebergs”
  - Because complicated caches, associativity, TLB, etc
  - Because register renaming and mu-ops and out-of-order
  - Because floating frequency and thermal throttling

- Modern CPUs are **very** complicated
- Do **not** predict (this was so Pentium III), **BENCH!**
- Still, a few simple rules’o’thumb **DO** exist!

CPU!

- Facts: it’s “always” **64-bit**, **multi-core**, **SIMD**, …

- **RULE OF THUMB : IT’S “ALWAYS” ~3 GHz**

- Fact: arithmetic is VERY cheap, **1 tick/op**… or less!
- Fact: but, beware of **“load-hit-store”!**
- How much “ops” for C = A + B * 3?

moving sideways!

CPU vs the “usual” ops

- Because who cares about C = A + B * 3
- Even at 5+ ticks that’s ~~zounds~~ billions

- And what are our **USUAL SUSPECTS**?

CPU vs the “usual” ops

- Okay, how MANY ticks (or nsecs) do we need for...

- `malloc()`?
- `qsort()`?
- `hash_find()`?
- `pthread_mutex_lock()/unlock()`?
- `fast_compress()/decompress()`?

CPU vs the “usual” ops

- Or for even higher level building blocks

- `accept()` etc overheads?
- `db_connect()` overheads?
- `json_parse()` cost?
- `protobuf::SerializeToString()` cost?
- …etc, **YOU** name it… and **BENCH** it

show me yours

- My code is mostly C++
- My “blocks” **are** malloc() etc
- My benchmarks reflect that

- YOUR code builds on them too

malloc()

- **RULE OF THUMB : ~1 M allocs/sec**

- Beware, you mileage may vary **GREATLY**
- **VERY** sensitive to size, ie 13 bytes vs 175,000 bytes
- **VERY** dependent on allocator, ie glibc vs je vs ...

- Note, **avoid** 1M allocs/sec! (Pools work fine in 2023)

qsort()

- **RULE OF THUMB : ~12…14 M ints/sec**
- Raw data = **0.70…0.85** sec / **10M** ints

- Homework #1, apply O(n log n) and 3 GHz, solve for C
- Homework #2, predict 1M or 100M timings
- Homework #3, compare with radix…

hash_find()

- **RULE OF THUMB : ~20…60 M calls/sec**

- K = short strings, 99.8% under 16 bytes
- V = ints, N = millions

- Implementations vary **SEVERELY**
- std::unordered_map << abseil ... << handmade

rwlock() readonly

- **RULE OF THUMB : ~5…10 M locks/sec**

- Raw data = **~33M** locks/sec, uncontended 1 rd
- Raw data = **~9M** locks/sec, 2 rd, 0 wr
- Raw data = **~6M** locks/sec, 128 rd, 0 wr

- Homework #1: bench this vs mutex()

raw data

rd	wr	pthreads	
1	0	30.5 ms	32.8 M/sec
2	0	230.8 ms	**8.7 M**/sec
4	0	630.2 ms	…
8	0	1227.7 ms	
16	0	2570.0 ms	
32	0	5090.7 ms	
64	0	11061.3 ms	
128	0	21830.8 ms	**5.8 M/sec**

rwlock() rdwr

- **RULE OF THUMB : ~3…6 M locks/sec**
- **RULE OF THUMB : ~2.5…0.1 M writes/sec**

- Different benchmark, **2x writers** at all times
- Readers slow down writers, as expected

raw data

rd	wr	pthreads	
1	2	739.8 ms	**4.0 M/sec**
2	2	1245.1 ms	**3.2 M/sec**
4	2	1428.0 ms	4.2 M/sec (!)
8	2	1831.3 ms	5.5 M/sec
16	2	3214.5 ms	5.6 M/sec
32	2	5678.9 ms	**6.0 M/sec (!!!)**
64	2	12132.8 ms	5.4 M/sec
128	2	23299.6 ms	5.6 M/sec

additions summary

malloc	~1.0	M/sec
qsort(int)	~12 … 14	M/sec
hash_find(short_str)	~20 … 60	M/sec
rwlock() reads	~3 … 6	M/sec
rwlock() writes	~0 … 2	M/sec
decompress()	~1 … 3	GB/sec
compress()	~0.1 … 1.0	GB/sec

compress()

- **RULE OF THUMB : ~1…3 GB/sec decompress**
- **RULE OF THUMB : ~0.1…1 GB/sec compress**

- Dean2010 => “Zippy” (now “Snappy”), still alive
- In 2023, we have **LZ4** and **zstd**

- Briefly: **LZ4** for speed, **zstd** for quality, and
- Briefly: NEVER gzip, NEVER Snappy

…that’s not quite CPU

the greatest beast


KINGSTON
FURY
FURY
DDR5
BEAST
KINGSTON
FURY
FURY
BEAST
DDR5

RAM!

- Maybe just my pet peeve, but IMO it’s RAM
- Seemingly simple, lots of timings, some docs, but…

- Predicting RAM perf?! **17th CIRCLE OF HELL**

- CPU = super complicated, but smth “works”
- RAM = seems innocuous, **NOTHING** “works”
  - Yes, that’s a rant

“Bad” news

- All those previous “CPU”benchmarks
- They are not CPU only
- They are **CPU + RAM**, all of them

- NB: they are ofc **VALID**
- You can’t “predict” those based on CPU only
- You must account for RAM... or bench (as we did)

“Good” news

- What do we do when the reality is too complicated?
- What if simple models don’t work?
- Easy, we use ML

- What do we do when theory just doesn’t compute?
- What if simple models don’t work?
- Easy, we use…

benchmarks

“Useful” news

- How to benchmark?
- You **MUST** account for cache
- You **MUST** have enough data… or it’s all cached

- L1 is **64 bytes**/cacheline, 16-32 KB total “today”
- L2 is **2…4 MB total** (per core) “today”
- Thus, 32 MB at the **VERY** least

Good news

- Aka statistics, the simplest form of ML ever
- Without further ado!

- **RULE OF THUMB : ~40…50M “io”/sec random**
- **RULE OF THUMB : ~15 GB/sec linear**

- **Everywhere!** 4th gen desktop, 11th gen laptop, Xeon…

Good news v.2

- But, ~~Apple M1 and~~ LPDDR5 changes everything

- **RULE OF THUMB : ~120…150M “io”/sec random**
- **RULE OF THUMB : ~40…50 GB/sec linear**

- Yes, that is **3 times** faster
- No, most servers are not there yet

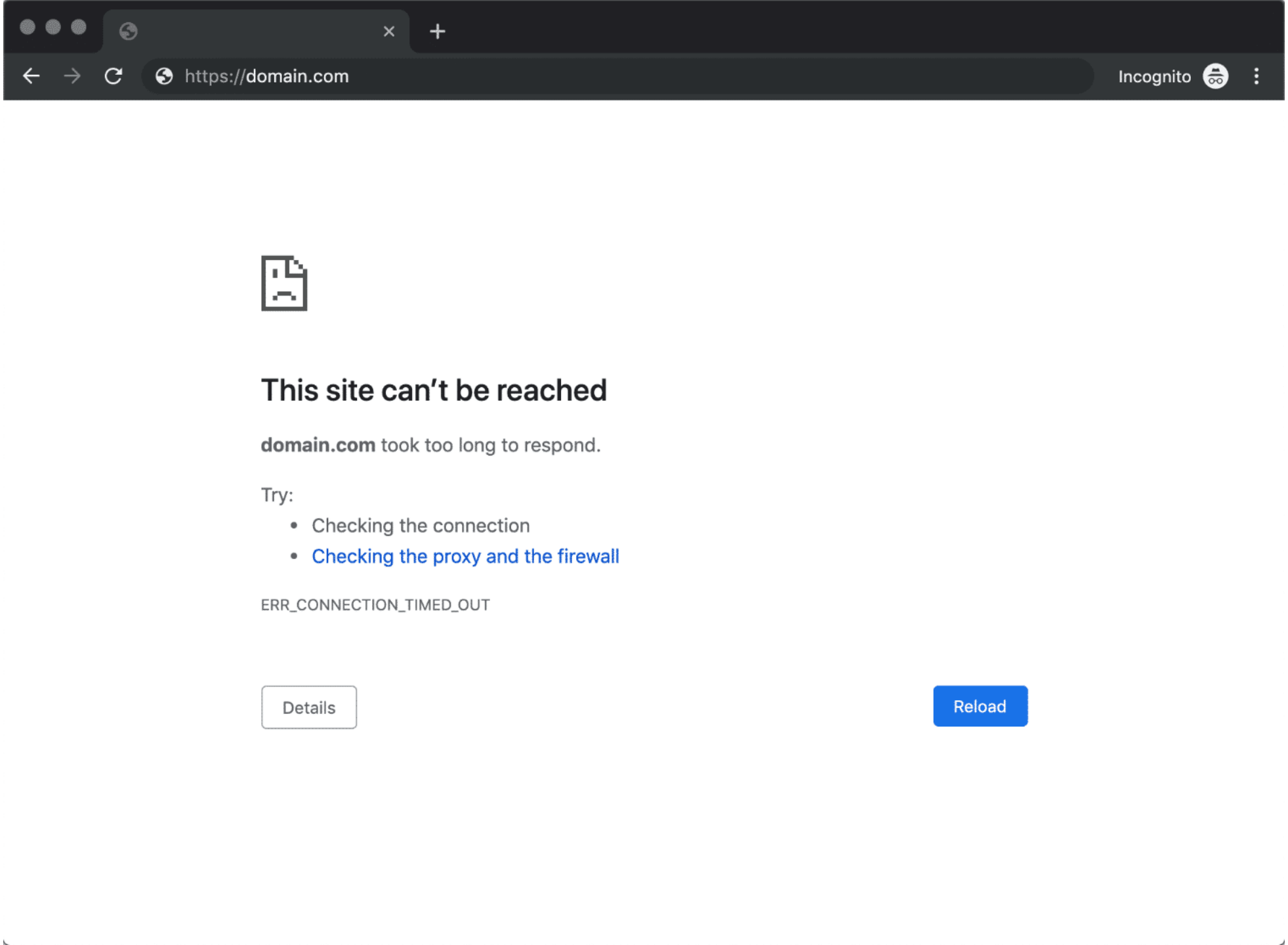
what have we learned?


SOUTH
PARK

Summary

- There are CPU, RAM, and disk
- There are [Dean2010] constants
- There are [Aks2022] constants (rules–of–thumb)
- You can (and should) bench **some of your own**

- You can **estimate system performance** with those
- And get reasonably precise results too, but


https://domain.com
Incognito
This site can’t be reached
domain.com took too long to respond.
Try:
Checking the connection
Checking the proxy and the firewall
ERR_CONNECTION_TIMED_OUT
Details
Reload

That’s it

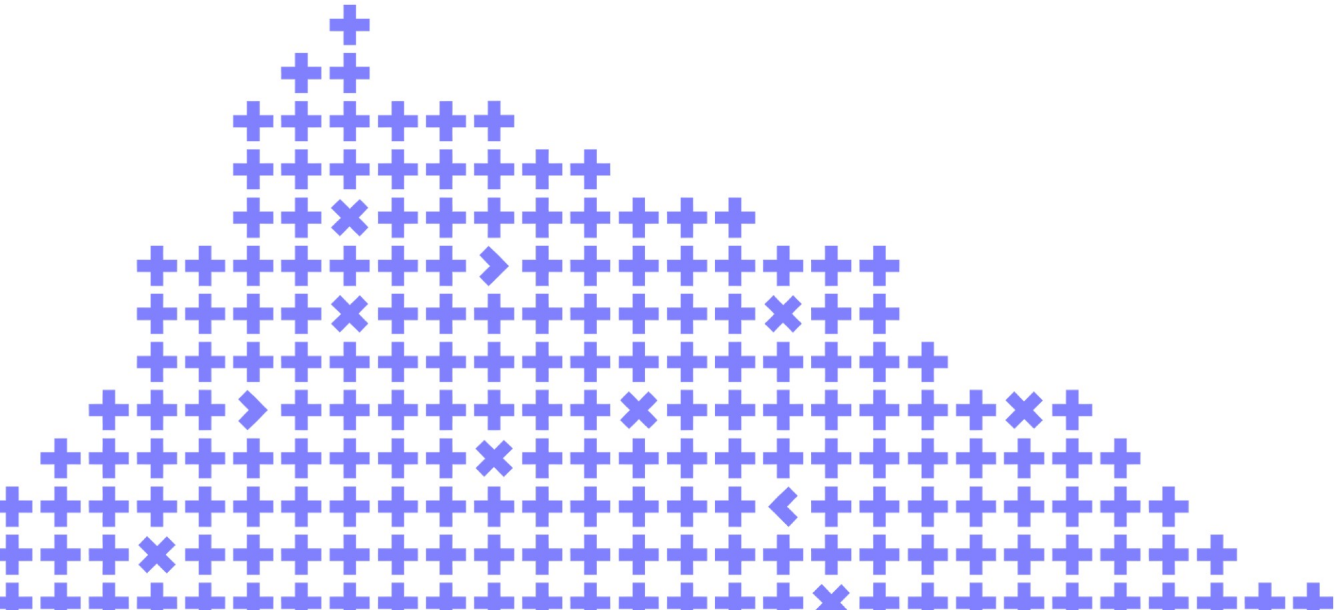
- Teaser #1, `csv_parse()`
- Teaser #2, `SELECT SUM(a+b*3)`

- Questions and comments?
- Homework and benchmarks?
- Very interesting, we want a workshop?
- Complex **and** boring, never again?

For the brave =>
telegram:// @shodanium

For the anonymous => QR

...or we could, you know,
TALK



High Load ++ Armenia

Co-organizer

Yandex

L1 cache reference	0.5 ns	
Branch mispredict	5 ns	
L2 cache reference	7 ns	14x L1 cache
Mutex lock/unlock	25 ns	
Main memory reference	100 ns	20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	3,000 ns	
Send 1K bytes over 1 Gbps network	10,000 ns	
Read 4K randomly from SSD*	150,000 ns	~1GB/sec SSD
Read 1 MB sequentially from memory	250,000 ns	
Round trip within same datacenter	500,000 ns	
Read 1 MB sequentially from SSD*	1,000,000 ns	~1GB/sec SSD, 4X memory
Disk seek	10,000,000 ns	20x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000 ns	80x memory, 20X SSD
Send packet CA->Netherlands->CA	150,000,000 ns	

dean-2010 oldies goldies

L1 cache reference	0.33 ns / 3000 Mops/s
Branch mispredict	???
hash_lookup()	16..50 ns / 20..60 Mops/sec
Main memory reference	20..25 ns / 40..50+ Mops/s
Rwlock (mutex?) lock/unlock	100..250 ns / 4..10 Mops/s
malloc()	1,000 ns / ~1M allocs/sec ???
[De]compress 1K bytes with lz4/zstd	???
Send 1K bytes over 1 Gbps network	10,000 ns / 100 MB/sec
Read 4K randomly from SSD *	20..200,000 ns / 5..50 Kops/sec
Read 1 MB sequentially from memory	66,000 ns / 15+ GB/sec
Read 1 MB sequentially from SSD *	333,000 ns / 3+ GB/sec
Round trip within same datacenter	500,000 ns / 0.5 ms
HDD disk seek	5..10,000,000 ns / 100..200 ops/sec
Read 1 MB sequentially from HDD	10,000,000 ns / 100+ MB/sec
qsort() 1M ints	70,000,000 ns / ~15 Mints/sec
Send packet CA->Netherlands->CA	150,000,000 ns / 150 ms

shodan-2022 update, TRUST NO ONE, INSERT YOUR VERY OWN BLOCKS HERE!

SUM(a+b*3) journey

- linear.cpp

```
int64_t r = 0; // sum(a+b*3)
for (int i = 0; i < N; i++) // N = 40M
  r += (rows[i].a + rows[i].b * 3); // row = 8 .. 64
b
```

- Sphinx
- MySQL
- ClickHouse

SUM(a+b*3) vs 64 bytes

```
linear.cpp, 64b, est RAM2010      0.63 sec
linear.cpp, 64b, est RAM2022      0.17 sec
linear.cpp, 64b, actual 6240R     0.19 sec

Sphinx, 64b, actual 40x           1.28 sec (!)
MySQL,  64b, actual 40x          10.72 sec
(!!!)
```

SUM(a+b*3) vs 8 bytes

```
linear.cpp, 8b, est RAM              0.022 sec
linear.cpp, 8b, est RAM+CPU          0.075..0.130
sec

linear.cpp, 8b, actual 6240R         0.024 sec
ClickHouse, 8b, actual 4x            0.020 sec (!!!)

WHY???
```

No magic there

1. **Automatic SIMD and unroll**, and pretty good ones
   => 4.5 instr/value, ~0.048 sec est

2. **Parallel execution, IPC > 1 (est 2.5)**
   => ??? actual ticks, ??? sec

3. **Parallel memory loads (read-ahead)**
   => max(ram,cpu) vs ram+cpu => ~0.022 sec

4. **Compression in ClickHouse**

4x SSE loop

```
01  movq        xmm0,xmm4 // this is 3.. i guess
02  movq        xmm1,mmword ptr [rcx+rax-8] // 1x unroll
03  pmulld      xmm1,xmm0
04  movq        xmm0,mmword ptr [rax-8]
05  paddd       xmm1,xmm0
06  pmovsxdq    xmm1,xmm1
07  paddq       xmm2,xmm1

08  movq        xmm0,xmm4
09  movq        xmm1,mmword ptr [rcx+rax] // 2x unroll
10  pmulld      xmm1,xmm0
11  movq        xmm0,mmword ptr [rax]
12  paddd       xmm1,xmm0
13  pmovsxdq    xmm1,xmm1
14  paddq       xmm1,xmm3
15  movdqa      xmm3,xmm1

16  lea         rax,[rax+10h] // add rax,16
17  sub         rdx,1
18  jne         00007FF6C4D71C51
```